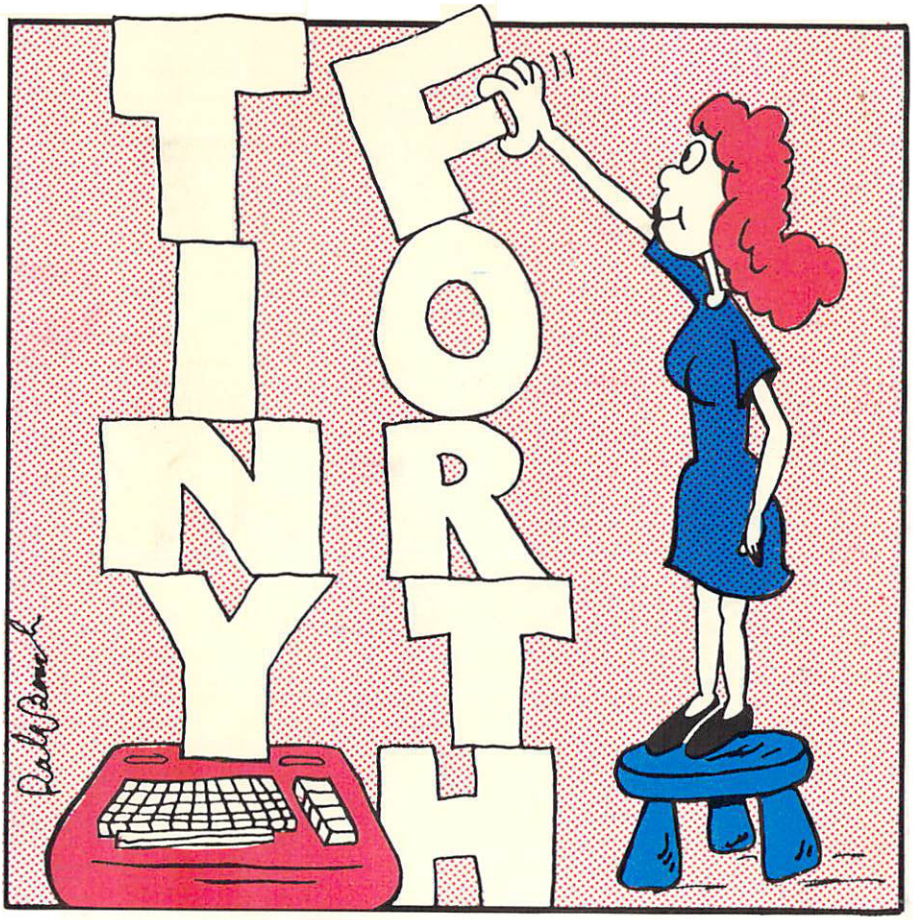

For the Commodore-64® or VIC-20®

KING MICROWARE Ltd.

==========

TINY FORTH

fig-Forth implementation for the

Commodore 64 and VIC-20

==========

Suite 210, 5950 Côte des Neiges, Montreal, Quebec H3S 1Z6

NOTE

This software is protected against copying. It is not possible to write to the program disk. In order to save the programs which you create with this software, you must use a new or previously initialized diskette. Below is a simple method of initializing a new disk.

```
OPEN15,8,15,"N0:DISKNAME,00"
```

It will take about 90 seconds, after which you may save your program to the new disk.

If the disk becomes unreadable within 30 days, we will replace it free of charge. After 30 days, we will replace it for a fee of $4.00..

WARRANTY

KING MICROWARE makes no warranties, expressed or implied, as to the fitness of this software package for a particular purpose. In no event will KING MICROWARE be liable for consequential damages. KING MICROWARE will replace any copy of this software which, is unreadable if returned within 30 days of purchase. Thereafter a nominal fee will be charged for replacement.

PREFACE

Nick Vrtis is a well known author of numerous articles and software that run on several popular 6502 micros. He's been writing both for many years now. Several months ago I casually mentioned to Nick that it would be nice to see FORTH running on the VIC-20. He responded with this version, which we call TINY FORTH. It's based on his SYM version which has been running for over two years. This new version runs not only on the VIC-20, but also on the COMMODORE 64.

TINY FORTH is based on the fig-FORTH release 1.1 for the 6502 dated September 1980. The original source listing and installation manual can be obtained from the Forth Interest Group, P.O. Box 1105, San Carlos, CA 94070. Nick has added several features to TINY FORTH that are not in the fig-FORTH version.

The primary changes made to the fig-FORTH are:

1. elimination of double precision (32 bit) arithmetic;
2. addition of a full screen editor and several editing words;
3. change the terminal buffer length to 88 from 84;
4. change the screen size to 40 character X 25 lines for the COMMODORE 64 or 22 character X 23 lines for the VIC-20;
5. elimination of the assembler.

TABLE OF CONTENTS

INTRODUCTION

FORTH is a high-level programming language. A program written in FORTH does not look at all like its equivalent written in BASIC. Nonetheless, FORTH is as equally capable as BASIC in problem solving.

In some respects FORTH is superior to BASIC. For instance, FORTH programs typically execute faster than BASIC programs. FORTH programs are usually shorter than BASIC programs. FORTH is "extensible" - you can add new words to the language. On the other hand, most people initially find that FORTH programs are cryptic compared to BASIC.

TINY FORTH is a version of FORTH for the COMMODORE 64 and VIC-20 microcomputers. It contains most of the language elements found in the fig-FORTH standard (most widely used version). TINY FORTH runs on a standard COMMODORE 64 or a VIC-20 with a minimum of an 8K memory expander.

This manual will show you how to get TINY FORTH up and running on your micro. It introduces you to the language elements that are included in TINY FORTH. However, it is not a tutorial on FORTH programming. For that, we recommend one of the references listed in the BIBLIOGRAPHY. Still we'll show you enough examples so that you are able to see how TINY FORTH works.

Our advice to you is to sit down with TINY FORTH and experiment with the words in the **GLOSSARY**. Not only will you find TINY FORTH a powerful language, but it is a fun one too. So have fun!

GETTING TINY FORTH SETUP

The distribution tape or diskette contains one the following:

NAME	CONTENTS
TF-20	TINY FORTH for the VIC-20
TF-64	TINY FORTH for the COMMODORE 64

Follow these directions to run TINY FORTH.

1. If using a VIC-20 version, make sure that you have an 8K memory expander installed. If not, turn off the computer, plug in the memory expander and then turn on the computer.

2a. If using a tape version of TINY FORTH, insert the distribution tape into the cassette drive. Make sure that the tape is completely rewound.

or

b. If using a disk version of TINY FORTH, carefully insert the distribution diskette into the disk drive and close the drive door.

3a. If using a tape version type:

LOAD "TF-20",1,1 or **LOAD "TF-64",1,1** and press the RETURN key. Be sure to type **,1,1.**

Press PLAY on the cassette recorder when asked to do so by the computer. With a COMMODORE 64, when the message **FOUND "TF-64"** appears on the screen, press the **C=** key to continue loading.

or

b. If using a disk version type:

LOAD "TF-20",8,1 or **LOAD "TF-64",8,1** and press the RETURN key. Be sure to type **,8,1.**

4. After TINY FORTH is loaded, the **READY.** prompt reappears on the screen. At this time, type **RUN** and press the RETURN key. A TINY FORTH signon message appears on the screen and a flashing cursor indicates that TINY FORTH is ready to accept your commands.

5. Now turn the the section called **SOME TINY FORTH BASICS.**

6. If you have to warm start TINY FORTH, you can type SYS6225 for a Commodore 64 or SYS8785 for a VIC-20.

SOME TINY FORTH BASICS

WORDS and the DICTIONARY

We talk to TINY FORTH using **WORDS.** All of TINY FORTH's words are stored in the **DICTIONARY.** TINY FORTH comes with a vocabulary of more than 175 words. You can also add new words to the dictionary, thereby extending your TINY FORTH language. We'll show you how to do this later.

A word consists of a character or several characters. However do not use any of the COMMODORE 64 or VIC-20 graphic characters for the name of a word. Some TINY FORTH words consist of special characters such as **+, -, *** or **/** which are similar to arithmetic operators of the BASIC language. A word is "executed" when the RETURN key (<RETURN>) is pressed.

STACK

The **STACK** is a place where TINY FORTH temporarily stores information. A TINY FORTH word usually expects the STACK to contain a certain number of values when they are executed. The word may also leave a value on the STACK as a result of its execution.

Values are placed on the STACK one at a time. The value that most recently placed on the STACK is called the top of the stack (TOS). If no values are left on the STACK, then it is said to be empty.

NUMBERS

TINY FORTH has many words that work with numbers. The numbers must be integers (whole numbers). They range in value from -32768 to +32767.

SPACES and the RETURN KEY

Unlike BASIC, the SPACE is a significant part of the TINY FORTH language. Unless a word is separated by a SPACE, TINY FORTH is not able to find the word in the dictionary.

The RETURN key (<RETURN>) tells TINY FORTH that you are ready to execute the word(s) that you have typed in. This lets you group together several words on a line (or even on multiple lines) and then execute them by pressing <RETURN>. The words are then executed in the order in which they were entered.

To introduce you to the language, here some examples:

A FIRST EXAMPLE

TINY FORTH considers numbers to be words. Whenever TINY FORTH encounters a number, it places that value on the stack. Remember that individual numbers are separated by spaces.

Let's begin by typing in the following words:

```
2 4 6 8 10 <RETURN>
```

Remember to leave a space between each number. After you press <RETURN>, TINY FORTH responds by printing **OK** meaning that it has placed the four number on the stack in the same order as you typed them in. First the number **2** is pushed onto the stack, making it the top of the stack (TOS). Next the number **4** is placed on the stack, pushing **2** down and leaving 4 at the TOS, etc. Finally after all of the words (numbers in this case) are executed, the stack looks like this:

```
TOS (top of stack)--->     10
                            8
                            6
                            4
bottom of stack------>      2
```

Now type the word . (a period, but commonly pronounced "dot") and press the RETURN key. TINY FORTH prints:

```
10
OK
```

The word . means print the value at the top of the stack and remove it from the stack. The stack therefore looks like this:

```
TOS (top of stack)--->     8
                           6
                           4
Bottom of stack------>     2
```

The value that was previously at the TOS (8) is gone. Now type the following:

```
. . . . <RETURN>
```

TINY FORTH will print:

```
8 6 4 2
OK
```

You told TINY FORTH to print the three values at the top of

the stack in succession. As each value at the TOS is printed, it is removed from the stack, leaving a new TOS value.

Now type . once more and <RETURN> and TINY FORTH prints:

0 MSG # 1

This means that the stack is empty since there are no further values on the stack. TINY FORTH keeps track of the bottom of the stack amd tells you if you are "bottoming out".

A SECOND EXAMPLE

The first example placed values on the stack and then removed them from the stack. The next example works a little differently.

Type:

25 3 <RETURN>. Don't forget to leave a space between the 25 and 3. The stack now looks like this:

TOS--------> 3
BOS--------> 25

Now type the word * <RETURN>. The word * tells TINY FORTH to multiply the value at the TOS by the value just underneath the TOS, remove both values and finally place the product at the TOS. After executing the word *, TINY FORTH places the value 75 at the TOS. To verify this, type . <RETURN> to print the TOS value.

In TINY FORTH, we describe the action of a word by showing the parameters on the stack before the execution of the word and the resulting values on the stack after the execution of a word.

For the multiply word * here is its description:

*** n1 n2 --- prod**

The values n1 and n2 are the multiplier and multiplicand. The --- indicates the execution point of the word *. Finally, **prod** is the resulting product which replaces both n1 and n2 on the stack.

The **GLOSSARY** in this manual is written using this description format.

A THIRD EXAMPLE

In the first example, the word . printed to the TOS value. However in doing so, it also removed it from the stack.

If we want to print the TOS without destroyint it, then we must first make a copy of the TOS. You can do this with the TINY FORTH word **DUP.**

For example the words:

```
5 DUP <RETURN>
```

would give a stack like this:

```
TOS--------> 5
BOS--------> 5
```

Printing using the . word, removes only the first value 5, leaving the second 5 at the TOS.

As previously mentioned, TINY FORTH comes with a dictionary filled with useful words. They are described in the glossary section of this manual. But you may also add your own words to TINY FORTH. To add words to TINY FORTH, you have to **COMPILE** them.

You tell TINY FORTH that you are creating a new word by using a : (colon). The : means that the name following the colon is the name of a new word. Any words following the name is part of the definition of that new word. Finally a ; (semicolon) tells TINY FORTH that the defintion is complete.

Using elements from the previous examples, you can create a new word called **SQR** (which prints the square of the number at the TOS). The new word and definition is compiled by typing:

```
: SQR DUP * . ; <RETURN>
CK
```

Notice that after typing the above line that TINY FORTH responds by printing CK. When TINY FOPTH is compiling (colon definition), the new word is not executed. The colon means that you are creating a new word, not executing it.

A closer look at the line:

:	beginning of definition (compile)
SQR	name of the new word (you could call it anything you wanted)
DUP	duplicate the value at the TOS
*	multiply the two values at the TOS leaving the product in their place

```
.       print the value at the TOS
;       end of definition (compile)
```

The name of the new word is arbitrary. You can call it SQUARE, SQRE or SQUARED. You can even use the name of a previously defined word. In this case, you will see a warning message MSG # 4 meaning that the new word is not unique. The most recent definition of the word is always executed. So if you redefine SQR at a later time, this last definition is executed when you type in SQR.

To try out the new word, type:

```
6 SQR <RETURN>
```

TINY FORTH responds by printing:

```
36
OK
```

The glossary description for the new word is:

```
SQR         n1 ---
```

The value n1 is the number that you want to square. **SQR** expects this number to be at the TOS before it is executed. No value is left on the stack after **SQR** is executed, so no values are shown the the right of the --- (execution point).

This concludes the "basics" section of TINY FORTH. For more information on other TINY FORTH words, consult the **GLOSSARY** or one of the references listed in the BIBLIOGRAPHY.

The next section describes the full screen editor. The EDITOR allows you to create **SCREEN**s which can be saved to disk or tape. A screen may contain new definitions or just plain text. The screen may later be **LOAD**ed (compiled from disk or tape) or **LIST**ed (displayed on screen).

THE EDITOR

The TINY FORTH editor works with a full screen at a time. To start out, type:

EMPTY-BUFFERS <RETURN>

which clears out the areas of memory (called **BUFFERS**) used by the editor. Now to edit a screen, type:

1 **:I** <RETURN> (no space between the : and I)

This tells TINY FORTH that you want to input data onto SCREEN number 1 (or another screen number). The cursor is positioned at the upper left hand corner of the screen and waits for you to input your data. You should first hold the shift key and press the CLR key in order to erase the screen of any data left on the screen.

With the editor, you can move the cursor anywhere on the screen using the cursor control keys. After the cursor is positioned where you want it, you can key in your data. The insert and delete keys work normally. Unlike the BASIC editor, the TINY FORTH editor works with a full screen at a time. What you see (on the screen) is what you get (in the **BUFFER**). It is not necessary to <RETURN> over each line to make changes.

After all of your changes are made to the screen, you may update the BUFFER or cancel the changes.

To update the BUFFER enter a SHIFTed <RETURN> (hold the SHIFT key while pressing the RETURN key). The contents of the screen are saved in the associated BUFFER, exactly as it appears on the screen.

To leave the editor without updating the BUFFER, press the RUN/STOP key instead. The contents of the BUFFER is left unchanged.

Commands and definitions keyed onto a screen using the editor are not immediately executed. If after editing a screen you want to execute the contents, type **1 LOAD** and TINY FORTH will execute the commands contained in screen buffer # 1 (of whatever screen number you were editing). Any colon definition contained in the screen buffer are compiled and if no errors are found they are entered into the dictionary. If errors are found, you can go back to re-edit the screen buffer.

To re-edit the screen, type **1 :E** <RETURN>. You may use the cursor control keys to position the cursor for making changes to the screen contents. Again, to save the contents of the screen to the BUFFER, press a SHIFTed <RETURN>, or

cancel these changes by pressing RUN/STOP.

After editing you will want to save the contents of the screen to disk or tape. To do this type:

1 PUT <RETURN>

The contents of the buffer (in this case buffer # 1) is saved to the disk or tape with the name **"SCREEN1"**.

After you become familiar with TINY FORTH, you will create many new words. Each time you use TINY FORTH, you must define these new words in the dictionary by typing them in (using colon definitions). An alternate way of doing this is to use the editor to create the source text for these words, and then save them to disk or tape (using PUT). Later, these screens are **LOAD**ed from the disk or tape thereby eliminating the need to retype them in. For example, to automatically compile any words on **"SCREEN1"**, you can type:

1 LOAD <RETURN>

TINY FORTH searches the buffers to find **"SCREEN1"**. If **"SCREEN1"** is not in memory, TINY FORTH searches the tape or disk for **"SCREEN1"** and when it finds it, reads the contents and executes the commands contained.

Below are the other editing words available in TINY FORTH. Try them out, since trying is the best way to learn to use TINY FORTH.

:E n ---
Full screen edit of screen # n. This word uses BLOCK to locate the requested screen, so it is read from tape or disk if necessary. Normal cursor editing is used. The TINY FORTH screen data is copied to the screen for editing. Note that the screen editor allows lines to wrap, but TINY FORTH treats the whole screen as one "line", and does not retain information concerning how lines are wrapped. Pressing the RUN/STOP key terminates editing without making any changes to the buffer. Pressing a shifted RETURN (hold SHIFT key while pressing <RETURN> key) exits the editor and saves the entire screen contents in the buffer.

:I n ---
Input from tape or disk to buffer as screen # n. Screen n must not already be in memory or error 5 results. Editing is the same as for :E. Note that the screen is not cleared at the start of :I, so any information on the screen is copied to the buffer unless you clear it.

:L addr n ---
Lists n screen lines from the buffer at addr. If the output device is other than the screen, a carriage return is output at the end of each screen line.

:M faddr taddr ---
Move a buffer from faddr to taddr. The length of the move is defined by B/BUF.

:O bfaddr n --- addr
Compute the offset of line n relative to bfaddr.

NOTES ON USING THE EDITOR

Unlike the BASIC editor, TINY FORTH works with a full screen at a time. When you are using :E or :I and enter a SHIFTed <RETURN> key, the entire screen contents are saved exactly as you see them. It is not necessary to <RETURN> over each line to update the buffer.

Using a tape version of TINY FORTH, it is possible to 'update' a tape if you are careful, since the screens are always the same length. The technique is to make sure the screen you want to update is in a buffer and update it (use the n :E command). Then use the following: **SCR @ 1 - GET.** You have to rewind the tape and then TINY FORTH reads in the screen just before the one you want to update. Make sure the tape is STOPped and enter **SCR @ PUT.** Make sure you press PLAY and RECORD when asked. Make sure that you reference the screen you want to save before the GET (via a 'n :E' or 'n BLOCK DROP') or GET may overlay the buffer you want to save.

THE SCREEN BUFFERS

As delivered the VIC-20 version of TINY FORTH has two buffers, while the COMMODORE 64 version has six buffers. Each buffer has a length equal to the length of a screen plus 2 bytes. These two bytes are used to hold the TINY FORTH screen number currently in that buffer. The number of buffers can be increased, but they must be contiguous, and each must be B/BUF + 2 bytes in length. NOTE that BLOCK-READ and BLOCK-WRITE do not use these buffers for their I/O. BLOCK-READ and BLOCK-WRITE are basically interfaces to the kernal read and write routines, and can be used to read/write any memory.

USEFUL EXAMPLES

To repeat, the easiest way to learn TINY FORTH is the try examples. Included below are several examples that explain some of the more useful words that will help you get the most from your TINY FORTH. Note that comments begin with **(** and end with **)**.

To change a tape version of TINY FORTH to a disk version, define a word called DISK as follows:

```
COMMODORE 64    : DISK BASE C@ HEX 8 1E38 C! BASE C! ;
VIC-20          : DISK BASE C@ HEX 8 2838 C! BASE C! ;
```

Here's the play by play description--

```
:               ( beginning of definition)
DISK            ( name of word)
BASE C@         ( leave current base on the stack)
HEX             ( change base to hexadecimal)
8 1E38 C!       ( change device or LOADing, PUTting,
   or                 GETting, etc. to device 8)
8 2838 C!
BASE C!         ( restore base to original)
```

To change a disk version of TINY FORTH to a tape version, define a word called TAPE as follows:

```
COMMODORE 64      : TAPE BASE C@ HEX 1 1E38 C! BASE C! ;
VIC-20            : TAPE BASE C@ HEX 1 2838 C! BASE C! ;
```

This word is similar to the word DISK above except that the device is 1 (tape device).

If you have a printer and want to get a hard copy of a screen you can define a word to do this.

```
: PLIST                        ( printer version of list)
  PRT 0 0 OPEN                 ( PRT  variable for printer
                                    with device 4
                                 0    no name
                                 0    length of name
                                 OPEN  opens logical file)
  PRT SETOUT                   ( set current output device
                                 to variable PRT)
  LIST                         ( perform normal LIST, but
                                 to device 4)
```

```
   0 SETOUT                    ( reset current output device
                                 to screen)
   PRT CLOSE ;                 ( close printer logical file)
```

Using PLIST, you can get a hard copy of any TINY FORTH screen(s). For example, if you have been editing screen number 2, you can get a hardcopy of it by typing:

```
     2  2  PLIST  <RETURN>
```

If you want to try a little animation then first define a word called DELAY.

```
:  DELAY  0  DO  1  DROP  LOOP ;
```

Next define a word called BIRD.

```
:  BIRD
  ."  v"                       ( clear screen)
   BEGIN
      ." [bird]"               ( draw bird)
      500 DELAY                ( wait a while)
      ." [bird]"               ( draw bird again)
      500 DELAY                ( wait some more)
      ."  [bird]"              ( erase bird)
      ?TERMINAL                ( check for RUN/STOP key)
    UNTIL                      ( repeat if not RUN/STOP)
    CR ;
```

Now type BIRD <RETURN> and watch some animation.

Here's how to make some noise with your computer. Using the word DELAY from the above example, you can define another word called UFO.

For the VIC-20:

```
:  UFO
   15 36878  C!                ( set volume to 15)
   254 130 DO                  ( vary the frequency from
                                 130 to 254)
             I  36876 C!       ( frequency to VIC chip)
             80 DELAY          ( wait a while)
          LOOP                 ( go back again)
   0 36878 C!                  ( turn off volume)
   0 36876 C! ;                ( turn off VIC chip)
```

For the COMMODORE 64:

```
: UFO
    15  54296  C!                ( volume)
    16  54276  C!                ( waveform triangular)
    34334 8583 DO                ( vary frequency
               I  54272 !          of voice 1)
               80 DELAY          ( wait a while)
            LOOP
    0  54276  C!                ( turn off volume)
    0  54296  C! ;               ( turn off waveform)
```

Now type in UFO <RETURN> to listen to the sound.

If you want to inspect the contents of memory (commonly known as dumping), in hexadecimal, use a word like this:

```
: DUMP

  SWAP 1+ SWAP              ( add 1 to ending memory addr)
  DO
     I  4   .R              ( print the addr)
     ."  @"                 ( print "@")
     I  5   +               ( prepare to print a line of
                              5 bytes of memory )
     I
     DO
       I  C@  3  .R         ( print one byte)
     LOOP                   ( now loop 5 times)
     CR                     ( new line)
     5                      ( add to address)
   +LOOP  ;                 ( next line has address 5 bytes
                              greater than last)
```

To dump a range of memory, say from $2000 thru $2080 you can use the following:

```
    HEX 2080 2000 DUMP <RETURN>
```

DUPLICATE WORD DEFINITIONS

To review, you can define a new word to TINY FORTH by using a colon definition to compile the word.

If you try to define a word with the same name as a previously defined word, then TINY FORTH displays a warning message - MSG # 4 meaning that the word is not unique.

In this case, TINY FORTH goes ahead and compiles the new word even though the word name is already in the dictionary. Any subsequent reference to a word by that name, refers to this last (most recent) definition.

You can remove words from the dictionary by using the word **FORGET.** **FORGET** cccc removes the word cccc and any other words that may have been defined since cccc was put into the dictionary. Let's look at this closer. Suppose we define the following words in this order. The periods (.....) represent the definition of each new word.

```
ENTRY 1          : aaaaa ..... ;
ENTRY 2          : ccccc ..... ;
ENTRY 3          : ddddd ..... ;
ENTRY 4          : eeeee ..... ;
ENTRY 5          : ccccc ..... ;
```

Note that entry 5 is a duplicate word, but TINY FORTH accepts it nonetheless. Using the word ccccc executes definition 5. If you type **FORGET ccccc**, then entry 5 is removed from the dictionary. Entries 1 thru 4 still remain in the dictionary. Next typing **FORGET ccccc** again removes entries from the dictionary. This time entries 2 thru 4 are removed leaving only entry 1. Typing ccccc gives MSG # 0 meaning that the word is not found in the dictionary.

FILENAMES

If you use LOAD, PUT, GET or LIST, TINY FORTH searches for files with the prefix **"SCREEN"**. For example, if you want to save screen 3 to a disk, then type **3 PUT** and the screen is written to the disk with the name **"SCREEN3"**. If you want to change the default name prefix (**"SCREEN"**), then use the following technique.

```
20  $STRING  NEWPREFIX          ( define string variable
                                    max. length of 20)
NEWPREFIX $" @:SCREEN"          ( move new name of prefix to
                                  new variable)
SNP NEWPREFIX $!                ( move new variable to TINY
                                  FORTH prefix)
```

Now any use of **LOAD, GET, PUT**, etc., will refer to the new prefix. For example using a new prefix of ABACUS, we can type:

```
2  PUT <RETURN>
```

saves the screen number 2 to a file named **"ABACUS2"**. Typing

```
8  LOAD <RETURN>
```

searches for a file named **"ABACUS8"** from the tape or disk (depending on the version in use).

GLOSSARY - NEW WORDS

$!

addrt addrf ---

Procedure to move a string pointed to by addrf to addrt. The length of the string pointed to by addrt determines the length of the move.

$"

Used in the form:

xxx $" ccccc"

Stores the string ccccc into the string variable xxx, setting the current length to the length of ccccc. There must be a blank after $". There must be at least one character (may be another blank) before the final ".

$<

$1 $2 --- f

Leave a true flag (non-zero) if string $1 is less than $2; otherwise leave a false flag.

$=

$1 $2 --- f

Leave a true flag (non-zero) if string $1 is equal to $2; otherwise leave a false flag.

$>

$1 $2 --- f

Leave a true flag(non-zero) if string $1 is greater than $2; otherwise leave a false flag.

$CMP

$1 $2 --- f

Procedure to compare string $1 and $2, leaving flag f as a result. f=0 if string $1 equals $2. f is positive and contains the position of first unequal character if string $1 is greater than $2. f is negative and contains the position of first unequal character if string $1 is less than $2.

$STRING

n ---

A defining word used in the form:

n $STRING cccc

to create a string variable named cccc with a parameter field ALLOTted n+1 bytes. The current string length is contained in the first byte of the parameter field. When cccc is later executed, the address of the parameter field (starting with the length byte) is left on the stack.

?IN

--- f

Test for non-standard input device. Leave false (zero) if output is set from keyboard. Leave device number otherwise.

?OUT --- f
Test for non-standard output device. Leave false (zero) if output is set for the screen. Leave the device number otherwise.

CLOSE addr ---
Close the ligical file number pointed to by addr (the same variable used to OPEN the file).

DEPTH --- count
Leave the number of 2-byte items on the stack.

FSN n --- addr l
Procedure to format the screen name for read/write operations. Uses the variable SNP to obtain the first portion of the name. Uses n as the screen number. This is converted to characters using the current BASE. Leaves the addr of where the name is built and the length (l). (l).

GET n ---
Read screen n into a buffer. Unlike BLOCK, this word forces a read of the screen from tape or disk. If the screen is currently in memory, that buffer is reused to hold the new copy of the screen. SCR is not updated by this word.

LM --- addr
Leave the address of the top of memory available for FCRTH to store dictionary words. Normally this is the same as FIRST.

MACHINE raddr addr ---
Procedure to jump to the subroutine at addr. The 6502 registers are loaded from the 4-bytes pointed to by raddr. The register contents are stored in the order X,Y,A,C. If C is non-zero, the carry flag is set. If C is zero, the carry flag is cleared. The subroutine must return via an RTS instruction. Registers and flags are returned in the 4-bytes pointed to by raddr. The full 6502 processor flags are returned in C. No registers need be saved by the user routine. Note that TINY FCRTH uses $4F-$51, $5A-$5B and $FB-$FE. These areas must be saved if your routine uses these areas.

OPEN addr naddr l ---
Procedure to open a logical file. The file parameters are pointed to by addr. The file parameter is as follows:

1-byte	logical file number
1-byte	device number
1-byte	secondary address

The name of the file is pointed to by naddr and l specified the name length. If there is no name,

then both of these parameters should be set to zero.

PRT --- addr
A user variable to hold the OPEN parameters for the printer. This variable is used by LIST to reset the printer logical file after using BLOCK to get a screen (because BLOCK may do a read). As delivered, the printer is defined as logical file 4, device 4, and secondary address 0 (upper case/graphics).

PUT n ---
Procedure to locate the buffer for screen n in memory (error 5, if not found) and write the buffer to tape or disk.

RSA --- addr
A user variable containing the pass registers to MACHINE.

SETIN addr ---
Procedure to set the current input device. The logical file must be previously OPENed. The logical file number is pointed to by addr (same variable used to OPEN the file). If addr is zero, then input is reset to the keyboard. This routine calls the Kernal routine CHKIN if addr is non-zero.

SETOUT addr ---
Procedure to set the current output device. The logical file must be previously OPENed. The logical file number is pointed to by addr (same variable used to OPEN the file). If addr is zero, then output will be reset to the screen. This routine calls the Kernal routine CHKOUT if addr is non-zero.

SNP --- addr
A user variable which contains the prefix used to create the screen name when a screen is saved or loaded. The first byte contains the number of characters in the name. There is enough room in the name for up to 16 characters. The name is suffixed by the screen number being read/written.

TOS --- n
Leave the initial length of the FORTH data stack.

UA --- addr
Leave the address of the start of the user variable area.

GLOSSARY - STANDARD WORDS

This glossary contains all of the word definitions in TINY FORTH. The definitions are presented in the order of their Ascii sort.

The first line of each entry shows a symbolic description of the action of the word (also called procedure) on the parameter stack. The symbols indicate the order in which input parameters have been placed on the stack. The dashes "---" indicate the execution point; any parameters left on the stack are listed afterwards. In this notation, the top of the stack is to the right.

The symbols include:

addr	memory address	
b	8 bit byte (i.e. hi 8 bits zero)	
c	7 bit ascii character (hi 9 bits zero)	
f	boolean flag.	zero = false, nonzero = true
ff	boolean false	flag = zero
n	16 bit signed integer number	
u	16 bit unsigned integer number	
tf	boolean true	flag = non-zero

Unless otherwise noted, all references to numbers are 16 bit signed integers. The high byte of the number is on top of the stack, with the sign in the leftmost bit.

All arithmetic is implicitly 16 bit signed integer math.

! n addr ---

Store 16 bits of n at address. Pronounced "store".

!CSP

Save the stack position in CSP. Used as part of the compiler security.

n1 --- n2

Generate from number n1, the next ascii character which is placed in an output string. Result n2 is the quotient after division by BASE, and is maintained for further processing. Used between <# and #>. See #S.

#> n1 --- addr count

Terminates numeric output conversion by dropping n1, leaving the text address and character count suitable for TYPE.

#S n1 --- n2

Generates ascii text in the text output buffer, by the use of #, until a zero number n2 results. Used between <# and #>.

' --- addr

Used in the form:

' nnnn

Leaves the parameter field address of dictionary word nnnn. As a compiler directive, executes a colon-definition to compile the address as a literal. If the word is not found after a search of CONTEXT and CURRENT, an appropriate error message is given. Pronounced "tick".

(

Used in the form:

(cccc)

Ignore a comment that will be delimited by a right parenthesis. May occur during execution or in a colon-definition. A blank after the leading parenthesis is required. At least one character must follow the blank.

(.")

The runtime procedure, compiled by ." which transmits the following in-line text to the selected output device. See ."

(+LOOP) n ---

The runtime procedure compiled by +LOOP, which increments the loop index by n and tests for loop completion. See +LOOP.

(ABORT)

Executes after an error. This word normally executes ABORT, but may be altered (with care) to a user's alternative procedure.

(DO)

The runtime procedure compiled by DO which moves the loop control parameters to the return stack. See DO.

(FIND) addr1 addr2 --- pfa b tf (ok)
addr1 addr2 --- ff (bad)

Searches the dictionary starting at the name field address addr2, matching to the text at address addr1. Returns parameter field address, length byte b of name field and boolean true for a good match. If no match is found, only a boolean false is left.

(LOOP)

The runtime procedure compiled by LOOP which increments the loop index and tests for loop completion. See LOOP.

(NUMBER) n1 addr1 --- n2 addr2
Convert the ascii text beginning at addr+1 with regard to BASE. The new value is accumulated into number n1, being left as n2. Addr2 is the address of the first unconvertable digit. Used by NUMBER.

* n1 n2 --- prod
Leave the signed product of two signed numbers.

+ n1 n2 --- sum
Leave the sum of n1 + n2.

+! n addr ---
Add n to the value at the address. Pronounced "plus-store".

+- n1 n2 --- n3
Apply the sign of n2 to n1, which is left as n3.

+BUF addr1 --- addr2 f
Advance the buffer address addr1 to the address of the next buffer addr2. Boolean f is false when addr2 is the buffer presently pointed to by variable PREV.

+LOOP n1 --- (run)
addr n2 --- (compile)
Used in a colon-definition in the form:
DO . . . n1 +LOOP
At runtime, +LOOP controls branching back to the corresponding DO base on n1, the loop index and the loop limit. The signed increment n1 is added to the index and the total compared to the limit. The branch back to DO occurs until the new index is equal to or greater than the limit (n1>0), or until the new index is equal to or less than the limit (n1<0). Upon exiting the loop, the parameters are discarded and execution continues ahead.

At compile time, +LOOP compiles the runtime word (+LOOP) and the branch offset computed from HERE to the address left on the stack by DO. n2 is used for compile time error checking.

+ORIGIN n --- addr
Leave the memory address relative by n to the origin of TINY FORTH. n is the minimum address unit, a byte. This definition is used to access or modify the boot-up parameters.

n ---
Store n into the next available dictionary memory cell, advancing the dictionary pointer. Pronounced "comma'.

- n1 n2 --- diff
Leave the difference of n1 - n2.

-->
Continue interpretation with the next disk or tape screen. Pronounced "next screen".

-DUP n1 --- n1 (if zero)
n1 --- n1 n1 (non-zero)
Reproduce n1 only if it is non-zero. This is usually used to copy a value just before IF, to eliminate the need for an ELSE part to drop it.

-FIND --- pfa b tf (found)
--- ff (non-found)
Accepts the next text word (delimited by blanks) in the input stream to HERE, and searches the CONTEXT and then CURRENT vocabularies for a matching entry. If found, the dictionary entry's parameter field address, its length byte b, and a boolean true is left. Otherwise, only a boolean false is left.

. n ---
Prints a number from a signed 16 bit two's complement value, converted according to the numeric BASE. A trailing blank follows. Pronounced "dot".

."
Used in the form:
." cccc"
Compiles an in-line string cccc delimited by a trailing ", with an execution procedure to transmit the text to the selected output device. If executed outside a definition, a ." will immediately print the text until the final ". There must be a blank after the ." . The blank is not printed. There must be at least one character (may be another blank) before the final ".

.R n1 n2 ---
Print the number n1 right aligned in a field whose width is n2. No following blank is printed.

/ n1 n2 --- quot
Leave the signed quotient of n1 / n2.

/MOD n1 n2 --- rem quot
Leave the remainder and signed quotient of n1 / n2. The remainder has the sign of the dividend.

0 1 2 3 --- n
These small numbers are used so often that it is attractive to define them by name in the dictionary as constants.

0<

n --- f
Leave a true flag if the number is less than zero (negative), otherwise leave a false flag.

0=

n --- f
Leave a true flag is the number is equal to zero, otherwise leave a false flag.

OBRANCH

f ---
The runtime procedure to conditionally branch. If f is false (zero), the following in-line parameter is added to the interpretive pointer to branch ahead or back. Compiled by IF, UNTIL, and WHILE.

1+

n1 --- n2
Leave n1 incremented by 1.

1-

n1 --- n2
Leave n1 decremented by 1.

2+

n1 --- n2
Leave n1 incremented by 2.

:

Used in the form called a colon-definition;
: cccc ... ;
Creates a dictionary entry defining cccc as equivalent to the following sequence of FORTH word definitions '...' until the next ';'. The compiling process is done by the text interpreter as long as STATE is non-zero. Other details are that the CONTEXT vocabulary is set to the CURRENT vocabulary and that words with the precedence bit set (P) are executed rather than being compiled.

;

Terminate a colon-definition and stop further compilation. Compiles the runtime ;S

;S

Stop interpretation of a screen. ;S is also the runtime word compiled at the end of a colon-definition which returns execution to the calling procedure.

<

n1 n2 --- f
Leave a true flag if n1 is less than n2; otherwise leave a false flag.

<#

Setup for pictured numeric output formatting using the words:
<# # #S SIGN #>
The conversion is done on a number, producing text at PAD.

<BUILDS

Used within a colon-definition:

```
: cccc <BUILDS ...
           DOES> ... ;
```

Each time cccc is executed, <BUILDS defines a new word with a high-level execution procedure. Executing cccc in the form:

```
cccc nnnn
```

uses <BUILDS to create a dictionary entry for nnnn with a call to the DOES> part for nnnn. When nnnn is later executed, it has the address of its parameter area on the stack and executes the words after DOES> in cccc. <BUILDS and DOES> allow runtime procedures to written in high-level rather than in assembler code.

= n1 n2 --- f

Leave a true flag if n1 = n2; otherwise leave a false flag.

> n1 n2 --- f

Leave a true flag if n1 is greater than n2; otherwise leave a false flag.

>R n ---

Remove a number from the computation stack and place as the most accessable on the return stack. Use should be balanced with R> in the same definition.

? addr ---

Print the value contained at the address in free format according to the current base.

?COMP

Issue error message if not compiling.

?CSP

Issue error message if stack position differs from value saved in CSP.

?ERROR f n ---

Issue an error message number n, if the boolean flag is true.

?EXEC

Issue an error message if not executing.

?LOADING

Issue an error message if not loading.

?PAIRS n1 n2 ---

Issue an error message if n1 does not equal n2. The message indicates that compiled conditionals do not match.

?STACK
Issue an error message if the stack is out of bounds.

?TERMINAL --- f
Perform a test of the terminal keyboard for the RUN/STOP key. A true flag indicates that it was pressed.

@ addr --- n
Leave the 16 bit contents of addr.

ABORT
Clear the stacks and enter the execution state. Redisplay the initial TINY FORTH prompt and return control to the operators terminal.

ABS n --- u
Leave the absolute value of n as u.

AGAIN addr n --- (compiling)
Used in a colon-definition in the form:
BEGIN ... AGAIN
At runtime, AGAIN forces execution to return to corresponding BEGIN. There is no errect on the stack. Execution cannot leave this loop (unless R> DROP is executed one level below).

At compile time, AGAIN compiles BRANCH with an offset from HERE to addr. n is used for compile-time error checking.

ALLOT n ---
Add the signed number n to the dictionary pointer DP. May be used to reserve dictionary space or re-origin memory.

AND n1 n2 --- n3
Leave the bitwise logical AND of n1 and n2 as n3.

B/BLK --- n
This constant leaves the number of bytes per buffer, they byte count read by BLOCK.

B/SCR --- n
This constatnt leaves the number of blocks per editing screen.

BACK addr ---
Calculate the backward branch offset from HERE to addr and compile into the next available dictionary memory address.

BASE --- addr
A user variable containing the current number base used for input and output conversion.

BEGIN --- addr n (compiling)
Occurs in a colon-definition in form:
BEGIN ... UNTIL
BEGIN ... AGAIN
BEGIN ... WHILE ... REPEAT
At runtime, BEGIN marks the start of a sequence that may be repetitively executed. It serves as a return point from the corresponding UNTIL, AGAIN or REPEAT. When executing UNTIL, a return to BEGIN will occur if the top of the stack is false; for AGAIN and REPEAT a return to BEGIN always occurs.

At compile time BEGIN leaves its return address and n for compiler error checking.

BL --- c
A constant that leaves the ascii value for a "blank" .

BLANKS addr count ---
Fill an area of memory beginning at addr with blanks for a length of count.

BLOCK n --- addr
Leave the memory addr of the block byte containing block n. If the block is not already in memory, it is tranferred from tape or disk to whichever buffer was most recently written.

BLOCK-READ addr1 addr2 count --- f
Read the file with the name pointed to by addr2 into the area starting at addr1. The file name length is specified by count. If count = 0, the and the device is tape, then the next file is read, regardless of name. f is the I/O error code.

BLOCK-WRITE addr1 addr2 addr3 count ---
Write memory to tape or disk. The memory written starts at addr1 and ends at addr2. addr3 points to the file name and has a file name length specified by count.

BRANCH
The runtime procedure to unconditionally branch. An inline offset isadded to the interpretive pointer IP to branch ahead or back. BRANCH is compile by ELSE, AGAIN, REPEAT.

BUFFER n --- addr
Obtain the next memory buffer, assigning it to block n. The addr left is the first address within the buffer for data storage. Updates PREV and USE.

C! b addr ---
Store 8 bits at addr.

C, b ---
Store 8 bits of b into the next available dictionary byte, advancing the dictionary pointer.

C@ addr --- b
Leave the 8 bit contents of memory address.

CFA pfa --- cfa
Convert the parameter field address of a definition to its code field address.

CMOVE addr1 addr2 count ---
Move the number of bytes specified by count beginning at addr1 to addr2. The contents of addr1 is moved first proceeding toward high memory.

COLD
The cold start procedure to adjust the dictionary pointer to the minimum startdard and restart via ABORT. May be called from the terminal to remove application programs and restart.

COMPILE
When the word containing COMPILE executers, the execution address of the word following COMPILE is copied (compiled) into the dictionary. This allows specific compilation situations to be handled in additionto simply compiling an execution address (which the interpreter already does).

CONSTANT n ---
A defining word used in the form:
n CONSTANT cccc
to create word cccc, with its parameter field containing n. When cccc is later executed, it will push the value of n to the stack.

CONTEXT --- addr
A user variable containing a pointer to the vocabulary within which dictionary searches will first begin.

COUNT addr1 --- addr2 n

Leave the byte address addr2 and byte count n of a message text beginning at address addr1. It is presumed that the first byte at addr1 contains the text byte count and the actual text starts with the second byte. Typically COUNT is followed by TYPE.

CR

Transmits a carriage return and line feed to the selected output device. Zeroes user variable OUT.

CREATE

A defining word used in the form:

```
CREATE  cccc
```

by such words as CODE and CONSTANT to create a dictionary header for a FORTH definition. The code field contains the address of the words parameter field. The new word is created in the CURRENT vocabulary.

CSP --- addr

A user variable temporarily storing the stack pointer position for compilation error checking.

DECIMAL

Sets the numeric conversion BASE for decimal input-output.

DEFINITIONS

Used in the form:

```
cccc  DEFINITIONS
```

Sets the CURRENT vocabulary to the CONTEXT vocabulary. In the example, executing vocabulary name cccc made in the CONTEXT vocabulary name cccc made it the CONTEXT vocabulary and executing DEFINITIONS made both specifify vocabulary cccc.

DIGIT

```
c  n1  ---  n2  tf      (ok)
c  n1  ---  ff          (bad)
```

Converts the ascii character c (using base n1) to its binary equivalent n2, accompanied by a true flag. If the onversion is invalid, leaves only a false flag.

DO

```
n1    n2  ---          (execute)
addr  n   ---          (compile)
```

Cccurs in a colon-definition in form:

```
DO   ...   LOOP
DO   ...   +LOOP
```

At runtime, DO begins a sequence with repetitive execution controlled by a loop limit n1 and an index with initial value n2. DO removes these from the stack. Upon reaching LOOP the index is incremented by one. Until the new index equals or

exceeds the limit, execution loops back to just after DO; otherwise the loop parameters are discarded and execution continues ahead. Both n1 and n2 are determined at runtime and may be the result of other operations. Within a loop 'I' will copy the current value of the index to the stack. See I, LOOP, +LOOP, LEAVE.

When compiling within the colon-definition, DO compiles (DO), leaves the following address addr and n for later error checking.

DOES>

A word which defines the runtime action within a high-level defining word. DOES> alters the code field and first parameter of the new word to execute the sequence of compiled word addresses following DOES>. Used in combination with <BUILDS. When the DOES> part executes, it begins with the address of the first parameter of the new word on the stack. This allows interpretation using this area or its contents. Typical uses include the FORTH assembler, multi-dimensional arrays and compiler generation.

DP --- addr

A user variable, the dictionary pointer, which contains the address of the next free memory above the dictionary. The value may be read by HERE and altered by ALLOT.

DROP n ---

Drop the number from the stack.

DUP n --- n n

Duplicate the value on the stack.

ELSE addr1 n1 --- addr2 n2 (compiling)

Occurs within a colon-definition in the form:

IF ... ELSE ... ENDIF

At runtime, ELSE executes after the true part following IF. ELSE forces execution to skip over the following false part and resumes execution after the ENDIF. It has no stack effect.

At compile-time ELSE replaces BRANCH reserving a branch offset, leaves the address addr2 and n2 for error testing. ELSE also resolves the pending forward branch from IF by calculating the offset from addr1 to HERE and storing at addr1.

EMIT c ---

Transmit ascii character c to the selected output device. OUT is incremented for each character output.

EMPTY-BUFFERS
Mark all block-buffers as empty by filling with zeroes. Updated blocks are not written to tape or disk. This is also an initialization procedure before first use of tape or disk.

ENCLOSE addr1 c --- addr1 n1 n2 n3
The text scanning primitive used by WORD. From the text address addr1 and an ascii delimiting character c, is determined by the byte offset to the first non-delimiter character n1, the offset to the first delimiter after the text n2, and the offset to the first character not included. This procedure will not process past an ascii 'null', treating it as an unconditional delimiter.

END
This is an 'alias' or duplicate definition for UNTIL.

ENDIF addr n --- (compile)
Occurs in a colon-definition in form:
IF ... ENDIF
IF ... ELSE ... ENDIF
At runtime, ENDIF serves only as the destination of a forward branch from IF or ELSE. It marks the conclusion of the conditional structure. THEN is another name for ENDIF. Both names are supported. See also IF and ELSE.

ERASE addr n ---
Clear a region of memory to zero from addr for n bytes.

ERROR line ---
Execute error notification and restart of system.

EXECUTE addr ---
Execute the definitionwhose code field address is on the stack. The code field address is also called the compilation address.

EXPECT addr count ---
Transfer characters from the terminal to addr, until a <RETURN> or the count of characters have been received. One or more nulls are added at the end of the text.

FENCE --- addr
A user variable containing an address below which FORGETting is trapped. To forget below this point, the user must alter the contents of FENCE.

FILL addr count b
Fill memory at the address with the specified count of bytes b.

FIRST --- addr
A constant that leaves the address of the first (lowest) block buffer.

FORGET
Executed in the form :
FORGET cccc
Deletes definition named cccc from the dictionary with all entries physically following it. In TINY FORTH, an error message occurs if the CURRENT and CONTEXT vocabularies are not currently the same.

FORTH
The name of the primary vocabulary. Execution make FORTH the CONTEXT vocabulary. Until additional user vocabularies are defined, new user definitions become a part of FORTH. FORTH is immediate, so it will execute during the creation of a colon-definition, to select this vocabulary at compile time.

HERE --- addr
Leave the address of the next available dictionary location.

HEX
Sets the numeric conversion base to sixteen (hexa-decimal)

HLD --- addr
A user variable that holds the address of the latest character of text during numeric output conversion.

HOLD c ---
Used between <# and #> to insert an ascii character c into a pictured numeric output string. e.g. 2E HOLD will place a decimal point.

I --- n
Used within a DO-LOOP to copy the loop index to the stack.

ID. addr ---
Print a definition's name from its name field address.

IF

f --- (runtime)
--- addr n (compile)

Occurs in a colon-definition in form:

IF (tp) ... ENDIF
IF (tp) ... ELSE (fp) ... ENDIF

At runtime, IF selects execution based on a boolean flag. If f is true (non-zero), execution continues ahead thru the true part. If f is false (zero), execution skips till just after ELSE to execute the false part. After either part, execution resumes after ENDIF. ELSE and its false part are optional; if missing, false execution skips to just after ENDIF.

At compile-time, IF compiles OBRANCH and reserves space for an offset at addr. addr and n are used later for resolution of the offset and error testing.

IMMEDIATE

Mark the most recently made definition so that when encountered at compile time, it will be executed rather than being compiled. i.e. the precedence bit in its header is set. This method allows definitions to handle unusual compiling situations, rather than build them into the fundamental compiler. The user may force compilation of an immediate definition by preceeding it with [COMPILE].

IN

--- addr

A user variable containing the byte offset within the current input text buffer from which the next text will be accepted. WORD uses and updates the value of IN.

INTERPRET

The outer text interpreter which executes or compiles text from the input stream depending on STATE. If the word name cannot be found after a search of CONTEXT and then CURRENT, it is converted to a number according to the current base. That also failing, an error message echoing the name with a " ?" will be given. Text input will be taken according to the convention for WORD.

KEY

--- c

Leave the ascii value of the next terminal key struck.

LATEST

--- addr

Leave the name field address of the topmost word in the CURRENT vocabulary.

LEAVE

Force termination of a DO-LOOP at the next opportunity by setting the loop limit equal to the current value of the index. The index itself remains unchanged, and execution proceeds normally until LOOP or +LOOP is encountered.

LFA pfa --- lfa

Convert the parameter field address of a dictionary definition to its link field address.

LIMIT --- addr

A constant leaving the address just above the highest memory available for a buffer.

LIST n1 n2 ---

Display the ascii text of screens n2 thru n1 on the selected output device. SCR contains the screen number during and after this process. Screens are read from tape or disk if not in memory.

LIT --- n

Within a colon-definition, LIT is automatically compiled before each 16 bit literal number encountered in input text. Later execution of LIT causes the contents of the next dictionary address to be pushed to the stack.

LITERAL n --- (compiling)

If compiling, then compile the stack value n as a 16 bit literal. This definition is immediate so that it will execute during a colon definition. The intended use is:

```
: xxx (calculate) LITERAL ;
```

Compilation is suspended for the compile time calculation of a value. Compilation is resumed and LITERAL compiles this value.

LOAD n ---

Begin interpretation of screen n. Loading will terminate at the end of the screen or at the ;S. See ;S or -->.

LOC-BLK n --- f

Locate block number n in memory only. PREV is updated to point to block if found, else it is unchanged. f is true if block is found. f is false if the block is not currently in the buffer.

LOOP addr n --- (compiling)
Occurs in a colon-definition in form:
DO ... LOOP
At runtime, LOOP selectively controls branching back to the corresponding DO based on the loop index and limit. The loop index is incremented by one and compared to the limit. The branch back to DO occurs until the index equals or exceeds the limit; at that time, the parameters are discarded and executioncontinues ahead.

At compile time, LOOP compiles (LOOP) and uses addr to calculate an offset to DO. n is used for error testing.

MAX n1 n2 --- max
Leave the greater of n1 and n2.

MESSAGE n ---
Print on the selected output device the text "MSG #" and the value of n according to the current BASE.

MIN n1 n2 --- min
Leave the smaller of two numbers.

MINUS n1 -- n2
Leave the two's complement of a number.

MOD n1 n2 --- mod
Leave the remainder of n1/n2, with the same sign as n1.

MON
Execution of 6502 BRK instruction.

MOVE addr1 addr2 n ---
Move the contents of n memory cells (16 bit contents) beginning at addr1 into n cells beginning at addr2. The contents of addr1 is moved first.

NFA pfa --- nfa
Convert the parameter field address of a definition to its name field.

NOP
A word which does nothing.

NUMBER addr --- n
Convert a character string left at addr with a preceeding count to a signed number n, using the current BASE. If numeric conversion is not possible, an error message is given.

OFFSET --- addr
A user variable which may contain a block offset to the disk buffer. The contents of OFFSET is added to the stack number by BLOCK.

OR n1 n2 --- or
Leave the bitwise logical or of two 16-bit values.

OUT --- addr
A user variable that contains a value incremented by EMIT. The user may alter and examine OUT to control display formatting.

OVER n1 n2 --- n1 n2 n1
Copy the second stack value, placing it as the new TOS.

PAD --- addr
Leave the address of the text output buffer, which is a fixed offset above HERE.

PFA nfa --- pfa
Convert the name field address of a compiled definition to its parameter field address.

PREV --- addr
A user variable containing the address of the buffer most recently referenced. Updated by BUFFER.

QUERY
Input 88 characters or text (or until a <RETURN> from the terminal. Text is positioned at the address contained in TIB with IN set to zero.

QUIT
Clear the return stack, stop compilation, and return control to the operators terminal. No message is given.

R --- n
Copy the top of the return stack to the computation stack.

R# --- addr
A user variable which may contain the location of an editing cursor, or other file related function.

R/W addr blk f ---
The TINY FORTH standard tape or disk read-write linkage. addr specified the source or destination block buffer, blk is the sequential number of the referenced block; and f is a flat for f=0 write and f=1 read. R/W determines the location on mass storage, performs the read-write and performs any error checking.

R> --- n
Remove the top value from the return stack and leave it on the computation stack. See >R and R.

R0 --- addr
A user variable containing the initial location of the return stack. Pronounced R-zero. See RP!

REPEAT addr n --- (compiling)
Used within a colon-definition in the form:
BEGIN ... WHILE ... REPEAT
At runtime, REPEAT forces an unconditional branch back to just after the corresponding BEGIN.

At compile time, REPEAT compiles BRANCH and the offset from HERE to addr. n is used for error testing.

ROT n1 n2 n3 --- n2 n3 n1
Rotate the top three values on the stack, bringing the third to the TOS.

RP!
Initialize the return stack pointer from user variable R0.

S0 --- addr
A user variable containing the initial value for the stack pointer. Pronounced S-zero. See SP!

SCR --- addr
A user variable containing the screen number most recently referenced by LIST.

SIGN n1 n2 --- n2
Stores an ascii "-" sign just before a converted numeric output string in the text output buffer when n is negative. n is discarded, but number n2 is maintained. Must be used between <# and #>.

SMUDGE
Used during word definition to toggle the "smudge bit" in a definitions' name field. This prevents an uncompleted definition from being found during dictionary searches, until compiling is completed without error.

SP!
Initialize the stack pointer from S0.

SP@ --- addr
A procedure to return the address of the stack position to the top of the stack, as it was before SP@ was executed.

SPACE

Transmit an ascii blank to the output device.

SPACES n ---

Transmit n ascii blanks to the output device.

STATE --- addr

A user variable containing the compilation state. A non-zero value indicates compilation.

SWAP n1 n2 --- n2 n1

Exchange the top two values on the stack.

TASK

A no-operation word which can mark the boundary between applications. By forgetting TASK and recompiling, an application can be discarded in its entirety.

THEN

An alias for ENDIF.

TIB --- addr

A user variable containing the address of the terminal input buffer.

TOGGLE addr b ---

Complement the contents of addr by the bit pattern b.

TRAVERSE addr1 n --- addr2

Move across the name field of a variable length name field. addr1 is the address of either the length byte or the last letter. If n=1, the motion is toward hi memory; if n=-1, the motion is toward low memory. The addr2 resulting is the address of the other end of the name.

TYPE addr count ---

Transmit count characters from addr to the selected output device.

U* u1 u2 --- ud

Leave the unsigned double number product of two unsigned numbers.

U/ ud u1 --- u2 u3

Leave the unsigned remainder u2 and unsigned quotient u3 from the unsigned double dividend and unsigned divisor d1.

UNTIL

f --- (runtime)
addr f --- (compile)

Occurs within a colon-definition in the form:

BEGIN ... UNTIL

At runtime, UNTIL controls the conditional branch back to the corresponding BEGIN. If f is false, execution returns to just after BEGIN; If true, execution continues ahead.

At compile-time, UNTIL compiles (OBRANCH) and an offset from HERE to addr. n is used for error tests.

USE

--- addr

A user variable containing the address of the buffer to use next as the least recently used. Updated by BUFFER.

UPDATE

Write the most recently referenced block (pointed to by PREV) to tape or disk.

USER

n ---

A defining word used in the form:

n USER cccc

which creates a user variable cccc. The parameter field of cccc contains n as a fixed offset relative to the user pointer register UP for this user variable. When cccc is later executed, it places the sum of its offset and the user area base address on the stack as the storage address of that particular variable.

VARIABLE

A defining wword used in the form:

n VARIABLE cccc

When VARIABLE is executed, it creates the definition cccc with its parameter field initialized to n. When cccc is later executed, the address of its parameter field (containing n) is left on the stack, so that a fetch or store may access this location.

VOC-LINK

--- addr

A user variable containing the address of a field in the definition of the most recently created vocabulary. All vocabulary names are linked by these fields to allow control for FORGETting thru multiple vocabularies.

VOCABULARY

A defining word used in the form:

```
VOCABULARY  cccc
```

to create a vocabulary definition cccc. Subsdequent use of cccc will make it the CONTEXT vocabulary which is searched first by INTERPRET. The sequence "cccc DEFINITIONS" will also make cccc the CURRENT vocabulary into which new definitions are placed.

In TINY FORTH, cccc will be so chained as to include all definitions of the vocabulary in which cccc is itself defined. All vocabularies ultimately chain to FORTH. By convention vocabulary names are to be declared IMMEDIATE. See VOC-LINK.

VLIST

List the names of the definitions in the context vocabulary. The RUN/STOP key will terminate the listing.

WARNING --- addr

A user variable containing a value controlling messages.

WHILE f --- (runtime)
addrl nl --- addrl nl addr2 n2

Occurs in a colon-definition in the form:

```
BEGIN ... WHILE (tp) ... REPEAT
```

At runtime, WHILE selects conditional execution based on boolean flag f. If f is true (non-zero), WHILE continues execution of the true part thru to REPEAT, which then branches back to BEGIN. If f is false (zero), execution skips to just after REPEAT, exiting the structure.

At compile time, WHILE emplaces (0BRANCH) and leaves addr2 of the reserved offset. The stack values will be resolved by repeat.

WIDTH --- addr

A user variable containing the maximum number of letters saved in the compilation of a definition's name. It must be 1 thru 31, with a default value of 31. The name, character count and its natural characters are saved, up the the value in WIDTH. The value may be changed at any time within the above limits.

WORD c ---

Red the next text characters from the input stream being interpreted, until a delimiter c is found, storing the packed character string beginning at the dictionary buffer HERE. WORD leaves the character count in the first byte, the characters, and the ends with two or more blanks. Leading occurrances of c are ignored. If BLK is zero, text is taken from the terminal input buffer, otherwise from the tape or disk block stored in BLK. See BLK, IN.

X

This is a pseudonym for the "null" of dictionary entry for a name of one character of ascii null. It is the exectuion procedure to terminate interpretation of a line of text from the terminal or within a tape or disk buffer, as both buffers always have a null at the end.

XOR n1 n2 --- xor

Leave the bitwise logical exclusive-or of two values.

[

Used in a colon-definition in form:

```
: xxx [ words ] more ;
```

Suspend compilation. The words after [are executed, not compiled. This allows calculation or compilation exceptions before resuming compilation with]. See LITERAL.

[COMPILE]

Used in a colon-definition in form:

```
: xxx [COMPILE] FORTH ;
```

[COMPILE] will force the compilationof an immediate definition, that would otherwise execute during compilation. The above example will select the FORTH vacabulary when xxx executes, rather than at compile time.

]

Resume compilation, to the completion of a colon-definition. See [.

MEMORY MAP

(all addresses are hexadecimal)

COMMODORE 64	VIC-20	DESCRIPTION
004F-0051	004F-0051	CODE FIELD POINTER
0057-005E	0057-005E	FORTH WORK AREA
00FB-00FC	00FB-00FC	NEXT FORTH WORD POINTER
0800	1200	BASIC STATEMENT WHICH IS THE SYS TO THE START OF TINY FORTH
080D	120D	FORTH STACK INDEX SAVEAREA. FOR **MACHINE**, THIS WILL POINT TO THE TOP OF THE FORTH STACK RELATIVE TO $080E
080E-08AB	120E-12AB	FORTH DATA STACK
08AD	12AD	INITIAL ADDRESS OF THE NAME FIELD OF THE LAST FORTH WORD ($20ED FOR C-64;$2AED FOR VIC-20)
08AF	12AF	POINTER TO TERNINAL INPUT BUFFER ($0200)
08B3	12B3	INITIAL **FORGET** FENCE (CAN'T **FORGET** WORDS BELOW THIS ($210B FOR C-64; $2B0B FOR VIC-20)
08B5	12B5	INITIAL ADDRESS OF START OF AVAILABLE MEMORY ($210B FOR C-64; $2B0B FOR VIC-20)
08B7	12B7	INITIAL VOCABULARY LINK POINTER ($178CFOR C-64;$2A8C FOR VIC-20)
0B2A	152A	**JSR CHROUT** AS PART OF **EMIT**
0B3B	153B	**JSR GETIN** AS PART OF **KEY**. NOTE THAT KEY USES THE **GET** ROUTINES, BUT WAITS FOR A CHARACTER TO BE ENTERED FROM THE KEYBOARD. THE CURSOR IS NOT ON AT THIS TIME, NOR IS THE CHARACTER ECHOED TO THE SCREEN. USE **HEX EAEA 0B3F !** FOR C-64 OR **HEX EAEA 15EF !** FOR THE VIC-20 TO CAUSE **KEY** TO LEAVE A ZERO ON THE STACK IF NO KEY IS DOWN. USE **HEX FAF0 0B3F !** FOR THE C-64 OR **HEX FAF0 15EF !** TO RESTORE THE DEFAULT CONDITION.
0B57	1557	**JSR STOP** AS PART OF **?TERMINAL**
0B7A	157A	**JSR CHROUT** AS PART OF CR
0F19	1919	**LM:** TOP OF AVAILABLE DICTIONARY MEMORY ($8844 FOR C-64; $3BC8 FOR VIC-20)
0F22	1299	UA: ADDRESS OF64-BYTEUSER VARIABLE AREA ($9FC0 FOR C-64; $3FC0 FOR VIC-20)
0F2E	192E	**LIMIT:** LIMIT OF MEMORY FOR BUFFERS ($9FC0 FOR C-64; $3FC0 FOR VIC-20)

0F3A	193A	**FIRST:** ADDRESS OF START OF BUFFER MEMORY ($8844 FOR C-64; $3BC8 FOR VIC-20)
0F45	1945	**ROWS:** NUMBER OF ROWS ON A SCREEN ($0019 FOR C-64; $0017 FOR VIC-20)
0F50	1950	**COLS:** NUMBER OF COLUMNS ON A SCREEN LINE ($0028 FOR C-64; $0016 FOR VIC-20)
0F5C	195C	**B/BUF:** BUFFER DATA LENGTH ($03E8 FOR C-64; $01FA FOR VIC-20)
1E38	2838	DEVICE NUMBER FOR SCREENSAVES AND LOADS ($01)
210B	2B0B	START OF AVAILABLE DICTIONARY MEMORY
8844-9FBF	3BC8-3FBF	BUFFER AREA
9FC0	3FC0	START OF 64 BYTE USER VARIABLE AREA (FIRST 44 BYTES ARE USED BY TINY FORTH)

The location and length of the USER AREA, screen buffers, and terminal buffer can be changed by changing the appropriate pointers as noted above. To allocate an extra buffer, use the following:

FIRST B/BUF 2+ - ' FIRST ! FIRST ' LM ! EMPTY-BUFFERS

Note that the change will be temporary unless you save a new copy of TINY FORTH. Be sureto enter the single quotes (called TIC) where shown.

SAVING A NEW VERSION OF TINY FORTH

You will probably extend your TINY FORTH vocabulary as you become more and more familiar with the language. Eventually you will want to create a new version of TINY FORTH which includes your personally customized words. To save a copy of TINY FORTH with new words and changes use the word **NEWFORTH.** NEWFORTH saves a new version of TINY FORTH to tape or disk with the name **NEW.TF-20** or **NEW.TF-64.** The following are the definitions for **NEWFORTH.**

For the COMMODORE 64

```
20  $STRING   NTF-64    ( THESE DEFINITIONS HAVE)
20  $STRING   TFDUMMY   ( ALREADY BEEN SET UP)
NTF-64  $" NEW.TF-64"   ( IN TINY FORTH)


: NEWFORTH
    TFDUMMY SNP $!          ( SAVE ORIGINAL FILENAME)
    SNP  NTF-64 $!          ( NAME OF NEW FILENAME)
    HEX LATEST 08AD !
    HERE 08B5 !
    VOC-LINK @ 08B7 !
    HERE 0800 SNP COUNT BLOCK-WRITE ( WRITE NEW FILE)
    SNP  TFDUMMY $! ;       ( RESTORE ORIGNAL FILENAME)
```

For the VIC-20:

```
20  $STRING   NTF-20    ( THESE DEFINITION HAVE)
20  $STRING   TFDUMMY   ( ALREADY BEEN SET UP)
NTF-20  $" NEW.TF-20"   ( IN TINY FORTH)


: NEWFORTH
    TFDUMMY SNP $!          ( SAVE ORIGINAL FILENAME)
    SNP  NTF-20 $!          ( NAME OF NEW FILENAME)
    HEX LATEST 12AD !
    HERE 12B5 !
    VOC-LINK @ 12B7 !
    HERE 1200 SNP COUNT BLOCK-WRITE ( WRITE NEW FILE)
    SNP  TFDUMMY $! ;       ( RESTORE ORIGNAL FILENAME)
```

ERROR MESSAGES

ERROR NUMBER	MEANING
0	WORD NOT FOUND
1	EMPTY STACK OR STACK OVERFLOW
2	DICTIONARY FULL
4	WARNING-NEW WORD IS NOT UNIQUE
5	REFERENCED SCREEN IS ALREADY IN MEMORY AND SHOULD NOT BE
9	ILLEGAL CHARACTER IN NAME
17	COMPILATION ONLY, USER IN DEFINITION
18	EXECUTION ONLY
19	CONDITIONALS NOT PAIRED
20	DEFINITION NOT FINISHED
21	IN PROTECTED DICTIONARY
22	USE ONLY WHEN LOADING
24	DECLARE VOCABULARY

I/O ERROR	MEANING
0	I/O ROUTINE TERMINATED BY RUN/STOP
1	TOO MANY OPEN FILES
2	FILE ALREADY OPEN
3	FILE NOT OPEN
4	FILE NOT FOUND
5	DEVICE NOT PRESENT
6	FILE IS NOT INPUT FILE
7	FILE IS NOT OUTPUT FILE
8	FILE NAME IS MISSING
9	ILLEGAL DEVICE NUMBER

BIBLIOGRAPHY

Here are some references in which you will find more about the FORTH language:

Ragsdale, William F., fig-FORTH INSTALLATION MANUAL, FORTH INTEREST GROUP, San Carlos, CA, 1980.

Brodie, Leo, STARTING FORTH, Prentice-Hall, Englewood Cliffs, 1981.

Knecht, Ken, INTRODUCTION TO FORTH, Howard W Sams & Co., Indianapolis, 1982.

Various, BYTE MAGAZINE, BYTE-MCGRAW HILL, August 1980 issue.

The contents of the GLOSSARY is taken in part from the publications provided by the Forth Interest Group.

TINY FORTH
PROGRAMMING LANGUAGE

FORTH is an exciting high-level programming language. The FORTH language is very different looking from BASIC. But in many ways it is more powerful than BASIC.

FORTH has a base vocabulary of simple words. A word is executed to perform a fundamental computing task. For example to add two numbers and display the sum using FORTH we would write:

```
106  75  +  .
```

These 'words' would add the numbers 106 and 75 together and display the result on the screen. To convert the decimal number 123 to hexadecimal and display the results we would write:

```
123 HEX .
```

which would display the result 7A on the screen.

Using FORTH we can create new and more powerful words by combining base words from the language such as

```
: DASH 45 EMIT ;
```

The word **DASH** will display a '-' onto the display. This new word in turn can be combined with others to create more complex words. By defining the word **DASHES** as follows -

```
: DASHES 0 DO DASH LOOP ;
```

we can cause any number of '-' to appear on the display by typing x DASHES where x is the number of dashes desired. Thus a single new word can do the word of dozens of fundamental words. We call FORTH an "extensible" language.